Sequenced Route Query with Semantic Hierarchy

Yuya Sasaki†, Yoshiharu Ishikawa‡, Yasuhiro Fujiwara §†, Makoto Onizuka†
†Graduate School of Information Science and Technology, Osaka University, Osaka, Japan
‡Graduate School of Information Science, Nagoya University, Nagoya, Japan
§NTT Software Innovation Center, Tokyo, Japan
sasaki@ist.osaka-u.ac.jp,ishikawa@i.nagoya-u.ac.jp,fujiwara.yasuhiro@lab.ntt.co.jp,onizuka@ist.osaka-u.ac.jp

ABSTRACT

The trip planning query searches for preferred routes starting from a given point through multiple Point-of-Interests (PoI) that match user requirements. Although previous studies have investigated trip planning queries, they lack flexibility for finding routes because all of them output routes that strictly match user requirements. We study trip planning queries that output multiple routes in a flexible manner. We propose a new type of query called *skyline sequenced route (SkySR)* query, which searches for all preferred sequenced routes to users by extending the shortest route search with the semantic similarity of PoIs in the route. Flexibility is achieved by the *semantic hierarchy* of the PoI category. We propose an efficient algorithm for the SkySR query, *bulk SkySR algorithm* that simultaneously searches for sequenced routes and prunes unnecessary routes effectively. Experimental evaluations show that the proposed approach significantly outperforms the existing approaches in terms of response time (up to four orders of magnitude). Moreover, we develop a prototype service that uses the SkySR query, and conduct a user test to evaluate its usefulness.

1 INTRODUCTION

Recently, technological advances in various devices, such as smart phones and automobile navigation systems, have allowed users to obtain real-time location information easily. This has triggered the development of location-based services such as Foursquare, which exploit rich location information to improve service quality. The users of the location-based services often want to find short routes that pass through multiple Points-of-Interest (PoIs); consequently, developing trip planning queries that can find the shortest routes that passes through user-specified categories has attracted considerable attention [4, 10]. If multiple PoI categories, e.g., restaurant and shopping mall, are in an ordered list (i.e., a *category sequence*), the trip planning query searches for a *sequenced route* that passes PoIs that match the user-specified categories in order.

Example 1.1. Figure 1 shows a road network with the following PoIs: "Asian restaurant", "Italian restaurant", "Gift shop", "Hobby shop", and "Arts&Entertainment (A&E)". Assume that a user wants to go to an Asian restaurant, an A&E place, and a gift shop in this order from start point v_q. The sequenced route query outputs route $R1$ because it is the shortest route from v_q that satisfied the user requirements ⟨Asian restaurant, A&E, gift shop⟩.

Existing approaches find the shortest route based on the user query. However, such approaches may find an unexpectedly long route because the found PoIs may be distant from the start point. A major problem with the existing approaches is that they only output routes that *perfectly* match the given categories [5, 14, 16]. To overcome this problem, we introduce flexible similarity matching based on PoI category classification to find shorter routes in a flexible manner. In the real-world, category classification often forms a *semantic hierarchy*, which we refer to as a *category tree*. For example, in Foursquare[1], the "Food" category tree includes "Asian restaurant," "Italian restaurant," and "Bakery" as subcategories, and the "Shop &Service" category includes "Gift shop," "Hobby shop," and "Clothing store" as subcategories (Figure 2). We employ this semantic hierarchy to evaluate routes in terms of two aspects, i.e., route length and the semantic similarity between the categories of the PoIs in the route and those specified in the user query. As a result, we can find effective sequenced routes that *semantically* match the user requirement based on the semantic hierarchy. For example, in Figure 1, route $R2$ satisfies the user requirement because it semantically matches the category sequence because Italian and Asian restaurants are in the same category tree. However, this approach may find a significantly large number of sequenced routes because the number of PoIs that flexibly match the given categories increases significantly. To reduce the number of routes to be output, we employ the skyline concept [2], i.e., we restrict ourselves to searching for the routes that are not worse than any other routes in terms of their scores (i.e., numerical values to evaluate the routes). Based on this concept, we propose the *skyline sequenced route (SkySR) query*, which applies the skyline concept to the route length and semantic similarity (i.e., we consider route length and semantic similarity as route scores). Given a start point and a sequence

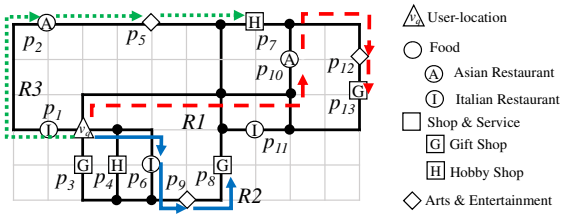


Figure 1: An example of a road network with PoIs

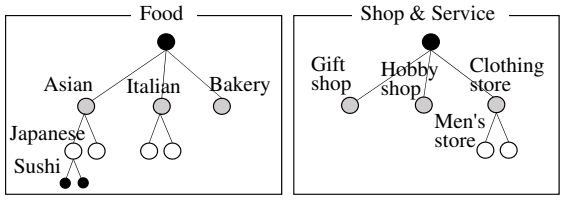


Figure 2: Examples of category trees in Foursquare

[1] https://developer.foursquare.com/categorytree

Table 1: Example routes in New York city

Approach	Distance	Sequenced route
Existing (e.g., [16])	3239 meters	Cupcake Shop → Art Museum → Jazz Club
Proposed	3239 meters	Cupcake Shop → Art Museum → Jazz Club
	1858 meters	Dessert Shop → Art Museum → Jazz Club
	1392 meters	Dessert Shop → Museum → Jazz Club
	823 meters	Dessert Shop → Museum → Music Venue

of PoI categories, a SkySR query searches for sequenced routes that are no worse than any other routes in terms of length and semantic similarity.

Example 1.2. Table 1 shows real-world examples of sequenced routes in New York city where a user plans to go to a cupcake shop, an art museum, and then a jazz club in this order. The existing approaches output a single route that matches the user's requirement perfectly. The proposed approach can output three additional routes that are shorter than the route found by the existing approach. Note that the additional routes also satisfy the user query semantically. The user can select a preferred route among all the four routes depending on how far he/she does not want to walk or their available time.

The SkySR query can provide effective trip plans; however, it incurs significant computational cost because a large number of routes can match the user requirement. Therefore, the SkySR query requires an efficient algorithm. The challenge is to search for SkySRs efficiently by reducing the search space without sacrificing the exactness of the result. We propose *bulk SkySR* algorithm (BSSR for short) that finds exact SkySRs efficiently. Recall that a feature of SkySRs is that their scores are no worse than those of other sequenced routes. BSSR exploits the branch-and-bound algorithm [9], which effectively prunes unnecessary routes based on the upper and lower bounds of route scores. In addition, to improve efficiency more, we employ four techniques to optimize BSSR. (1) First, we initially find sequenced routes to calculate the upper bound. (2) We tighten the upper bound by arranging the priority queue and (3) tighten the lower bound by introducing minimum distances. (4) we keep intermediate results for later processing, which refer to as *on-the-fly caching*. Our approach significantly outperforms existing approaches in terms of response time (up to four orders of magnitude) without increasing memory usage or sacrificing the exactness of the result.

The main contributions of this paper are as follows.

- We introduce a semantic hierarchy to the route search query, which allows us to search for routes flexibly.
- We propose the *skyline sequenced route (SkySR) query*, which finds all preferred routes related to a specified category sequence with a semantic hierarchy (Section 4).
- We propose an exact and efficient algorithm and its optimization techniques to process SkySR queries (Section 5).
- We discuss variations and extensions of the SkySR query. The SkySR query can be applied to various user requirements and environments (Section 6).
- We demonstrate that the proposed approach works well in terms of response time and memory usage by performing extensive experiments. (Section 7).
- We develop a prototype service that employs the SkySR query and conduct a user test to evaluate usefulness of the SkySR query. (Section 8).

The remainder of this paper is organized as follows. Section 2 introduces related work. Section 3 describes the problem formulation, and Section 4 defines the SkySR query. Section 5 presents the proposed algorithm. In Section 6, we discuss variations and extensions of the SkySR query. Sections 7 and 8 present experiment and user test results, respectively, and Section 9 concludes the paper.

2 RELATED WORK

First, we review trip planning query studies related to the SkySR query. Then, we review some studies related to the skyline operator. To the best of our knowledge, no study has considered a skyline sequenced route; thus, our problem cannot be solved efficiently using existing approaches.

Trip planning: We categorize trip planning queries in Table 2. Note that all existing trip planning queries only output routes that perfectly match the user-specified category sequences. Moreover, since most trip planning queries assume Euclidean distance, they cannot find SkySRs, in which road network distance is assumed. Dai et al. [4] proposed a personalized sequenced route and assumed that PoIs have ratings as well as categories and that users assign weighting factors as preferences. Although this personalized sequenced route considers route lengths and ratings, it only outputs the route that perfectly matches the given categories and has the best score based on lengths, ratings, and preferences. Only the optimal sequenced route (OSR) is applicable to find SkySRs without modification because the OSR and SkySR are based on the same settings (except for scoring). Sharifzadeh et al. [16] proposed two algorithms to find OSRs in road networks: the *Dijkstra-based solution* and the *Progressive Neighbor Exploration (PNE) approach*. The main difference between these algorithms is that the Dijkstra-based solution employs the Dijkstra algorithm to search for PoIs and the PNE approach employs the nearest neighbor search. It has been reported that these algorithms are comparable in terms of performance [16]. Thus, we consider both algorithms to verify the performance of the proposed approach.

Skyline: The skyline operator was proposed previously [2]. Few studies have considered the skyline concept for route searches. Recently, the skyline route (or skyline path) has received considerable attention [1, 6, 8, 13, 17, 18, 20]. A skyline route assumes that edges on road networks are associated with multiple costs, such as distance, travel time, and tolls. Here, the objective is to find skyline routes from a start point to a destination considering these multiple costs. However, since we specify a category sequence rather than a destination, we cannot apply conventional algorithms to find SkySRs. The continuous skyline query in road networks (e.g., [7]) searches for the skyline PoIs for a moving object considering both the PoI category and the distances to the moving object. Because continuous skyline queries search for a single PoI category, these solutions are not applicable to SkySR queries, which obtain routes that pass through multiple PoIs.

3 PRELIMINARIES

Table 3 summarizes the notations used in this paper. We assume a connected graph $G = (\mathbb{V} \cup \mathbb{P}, \mathbb{E})$, where $\mathbb{V}$, $\mathbb{P}$, and $\mathbb{E} \subseteq (\mathbb{V} \cup \mathbb{P}) \times (\mathbb{V} \cup \mathbb{P})$ represent the sets of vertices, PoI vertices, and edges, respectively. This graph corresponds to a road network that contains PoIs. The numbers of vertices, PoI vertices, and edges are denoted $|\mathbb{V}|$, $|\mathbb{P}|$, and $|\mathbb{E}|$, respectively. PoI vertex $p \in \mathbb{P}$ is associated with category $c \in \mathbb{C}$, where $\mathbb{C}$ is the set of categories. We denote the category of PoI vertex p as c_p, and assume that

Table 2: Types of trip planning queries.

Type	Distance metrics	Order	Destination	Result	Scores
SkySR (proposed)	Network	Total	Yes or No	Exact	Length and semantic
Optimal sequenced route (OSR) [16]	Euclidean or Network	Total	Yes or No	Exact	Length
Sequenced route [5, 14]	Network	Total	Yes	Exact	Length
Personalized sequenced route [4]	Euclidean	Total	No	Approximate	Length and rating
Trip planning [10]	Euclidean or Network	Non	Yes	Approximate	Length
Multi rule partial sequenced route [3]	Euclidean	Partial	No	Approximate	Length
Multi rule partial sequenced route [11]	Euclidean	Partial	No	Exact	Length
Multi-type nearest neighbor [12]	Euclidean	Non	No	Exact	Length

Table 3: Notations

Symbol	Meaning
$\mathbb{V}$	Set of vertices
$\mathbb{P}$	Set of PoI vertices
$\mathbb{E}$	Set of edges
p	PoI vertex
$\mathbb{C}$	Set of categories
c	Category
t	Category tree
c_p	Category of PoI vertex p
t_c	Category tree of c
$\mathbb{P}_c$	Set of PoI vertices associated with c
$\mathbb{P}_t$	Set of PoI vertices associated with t
$\mathbf{S}$	Category sequence (sequence of categories)
$\mathbf{R}$	Route (sequence of PoI vertices)
$\mathbf{S}_R$	Sequential PoI categories in $\mathbf{R}$
$l(\mathbf{R})$	Length score of $\mathbf{R}$
$s(\mathbf{R})$	Semantic score of $\mathbf{R}$
$\mathcal{R}$	Set of routes
$\mathcal{E}(\mathbf{R})$	Set of super-routes of $\mathbf{R}$
$\mathcal{S}$	Minimal set of sequenced routes
$\mathbf{S}_q$	Category sequence specified by user
v_q	Start point specified by user

each PoI is associated with a single category. Each category is associated with category tree t, and we denote the category tree of category c as t_c. We denote the set of PoI vertices associated with c and the set of PoI vertices associated with the category tree t as $\mathbb{P}_c$ and $\mathbb{P}_t$, respectively. If a PoI vertex is associated with category c, it is also associated with all ancestor categories of c in t_c. Each edge $e(u_i, u_j)$ in $\mathbb{E}$ is associated with a weight $w(u_i, u_j)$ (≥ 0). The weight can represent either travel duration or distance. Next, we define several terms required to introduce the skyline sequenced route (SkySR).

Definition 3.1. **(Category sequence)** A *category sequence* $\mathbf{S} = \langle c_S[1], c_S[2], \ldots, c_S[|\mathbf{S}|]\rangle$ is a sequence of categories, where $|\mathbf{S}|$ is the size of $\mathbf{S}$. $c_S[i] \in \mathbb{C}$ denotes the i-th category in $\mathbf{S}$. A *super-category sequence* of $\mathbf{S}$ is a category sequence where each i-th category is either $c_S[i]$ or an ancestor of $c_S[i]$ ($1 \leq i \leq |\mathbf{S}|$) in the category tree.

Definition 3.2. **(Route)** A *route* $\mathbf{R} = \langle p_R[1], \ldots, p_R[|\mathbf{R}|]\rangle$ is a sequence of PoI vertices in a road network, where $p_R[i] \in \mathbb{P}$ and $|\mathbf{R}|$ denote the i-th PoI vertex in $\mathbf{R}$ and the size of $\mathbf{R}$, respectively. $\mathbf{S}_R$ denotes the category sequence of $\mathbf{R}$ (i.e., $\langle c_{p_R[1]}, \ldots, c_{p_R[|\mathbf{R}|]}\rangle$). In addition, we define a *super-route* of $\mathbf{R}$ as an extended route of $\mathbf{R}$, such as $\langle \mathbf{R}, p_i, p_j, \ldots\rangle$. In other words, a super-route of $\mathbf{R}$ is obtained by adding a sequence of PoI vertices to the end of $\mathbf{R}$. $\mathcal{R}$ and $\mathcal{E}(\mathbf{R})$ denote a set of routes and a set of super-routes of $\mathbf{R}$, respectively. Moreover, given a route $\mathbf{R} = \langle p_R[1], \ldots, p_R[|\mathbf{R}|]\rangle$ and a PoI vertex p, we define $\mathbf{R} \oplus p = \langle p_R[1], \ldots, p_R[|\mathbf{R}|], p\rangle$.

Definition 3.3. **(Category similarity)** Given two categories c and c', the similarity $sim(c, c') \in [0, 1]$ is calculated by an arbitrary function such as the *Wu and Palmer similarity* or path length [15, 19]. We assume the following relations in the similarity.

- c is irrelevant to c' if both exist in different category trees; thus, we obtain $sim(c, c') = 0$.
- c *semantically matches* c' if c and c' are in the same category tree; thus, we obtain $0 < sim(c, c') \leq 1$.
- c *perfectly matches* c' if c and c' are the same; thus, we obtain $sim(c, c') = 1$.

Note that a semantic match subsumes a perfect match.

We define a *sequenced route* using the above definitions. The difference between our definition of *sequenced route* and the previous definition [16] is that we consider category similarity.

Definition 3.4. **(Sequenced route)** Given category sequence $\mathbf{S} = \langle c_S[1], \ldots, c_S[|\mathbf{S}|]\rangle$, $\mathbf{R} = \langle p_R[1], \ldots, p_R[|\mathbf{R}|]\rangle$ is *a sequenced route* of category sequence $\mathbf{S}$ if and only if it satisfies (i) $|\mathbf{R}| = |\mathbf{S}|$, (ii) $c_S[i]$ semantically matches $c_{p_R[i]}$ for all i such that $1 \leq i \leq |\mathbf{S}|$, and (iii) all PoI vertices in $\mathbf{R}$ differ each other.

Definition 3.5. **(Route scores)** Given category sequence $\mathbf{S}$ and vertex v as a start point, we define two scores for route $\mathbf{R}$: *length score* $l(\mathbf{R}) \in [0, \inf]$ and *semantic score* $s(\mathbf{R}) \in [0, 1]$. We define the length score $l(\mathbf{R})$ as follows:

$$l(\mathbf{R}) = D(v, p_R[1]) + \Sigma_{i=1}^{|\mathbf{R}|-1} D(p_R[i], p_R[i+1]), \quad (1)$$

where $D(u_i, u_j)$ denotes the smallest weight sum of the edges on the routes between vertices (or PoIs) u_i and u_j. The semantic score $s(\mathbf{R})$ is calculated by an aggregation function f as follows:

$$s(\mathbf{R}) = f(h_1, h_2, \ldots, h_{|\mathbf{R}|}), \quad (2)$$

where h_i denotes $sim(c_S[i], c_{p_R[i]})$. We assume that, if all $h_i = 1$, $s(\mathbf{R}) = 0$, i.e., if all PoI vertices in a route perfectly match the categories, the semantic score of the given route is 0. We also assume that $\underline{s}(\mathbf{R})$ is the possible minimum semantic score of $\mathbf{R}$ when it is a sequenced route. Without loss of generality, preferred routes have small length and semantic score.

4 THE SKYLINE SEQUENCED ROUTE QUERY

Here, we define the SkySR query. Intuitively, a SkySR is a potential route that may be the best route related to the user's requirement. A potential route is a route that is not *dominated* by any other routes; the notion of *dominance* is used in the *skyline operator* [2]. We define dominance for sequenced routes and SkySR query in the following.

Definition 4.1. **(Dominance)** Let $\mathcal{R}$ be the set of all sequenced routes starting from point v for category sequence $\mathbf{S}$. For two sequenced routes $\mathbf{R}, \mathbf{R}' \in \mathcal{R}$, we say that $\mathbf{R}$ dominates $\mathbf{R}'$ if we have (i) $l(\mathbf{R}) < l(\mathbf{R}')$ and $s(\mathbf{R}) \leq s(\mathbf{R}')$ or (ii) $s(\mathbf{R}) < s(\mathbf{R}')$ and $l(\mathbf{R}) \leq l(\mathbf{R}')$. If two sequenced routes have the same length and

semantic scores, the routes are *equivalent* in the dominance, and a set of sequenced routes is *minimal* if it has no equivalent routes.

Definition 4.2. **(SkySR query)** Given vertex v_q as a start point and category sequence $\mathbf{S}_q$, a skyline sequenced route is a sequenced route not dominated by other routes. Let $\mathcal{R}$ be the set of all sequenced routes from start point v_q for category sequence $\mathbf{S}_q$, and let $\mathcal{S}$ be a *minimal* set of the sequenced routes. The SkySR query returns $\mathcal{S}$ that includes sequenced routes such that all $\mathbf{R} \in \mathcal{S}$ are SkySRs and all $\mathbf{R}' \in \mathcal{R} \setminus \mathcal{S}$ are dominated by or equivalent to some of $\mathbf{R} \in \mathcal{S}$.

An naive solution to find SkySRs is to first enumerate SkySR candidates by iteratively executing OSR queries for any super-category sequences of $\mathbf{S}_q$ and then check the dominance among the routes. The number of super-category sequences of $\mathbf{S}_q$ increases exponentially as the depth of the category in the category tree and the size of $\mathbf{S}_q$ increase. Thus, although OSR algorithms can find a sequenced route efficiently, we must repeat many searches. As a result, the naive solution needs significantly high computational cost to find SkySRs.

5 PROPOSED ALGORITHM

In this section, we present the proposed approach, which we refer to as the *bulk SkySR algorithm* (BSSR), that finds SkySRs efficiently. Section 5.1 presents the BSSR design policy, and Section 5.2 explains the BSSR procedure. In Section 5.3, we propose optimization techniques for BSSR. We also theoretically analyze its performance in Section 5.4. Finally, we show a running example of BSSR in Section 5.5. In Section 5, we assume undirected graphs in which each PoI vertex is associated with only one category and that users give sequences of single PoI categories. However, in a real application, the graphs would be directed graphs, each PoI vertex would be associated with multiple categories, and users may specify complex categories. Section 6 describes how we handle the above conditions.

5.1 Design Policy

Our idea to improve efficiency is to find sequenced routes simultaneously (i.e., by searching sequenced routes in bulk) in order to reduce the search space. We have two choice as the basis for our approach; Dijkstra-based or nearest neighbor-based approaches [16]. We use the Dijkstra-based approach as the basis of our algorithm. Recall that a SkySR query has two scores for a route, i.e., length and semantic scores. To find all SkySRs, we must find routes that have small category scores even if the routes have large length scores. However, PoIs that are included in the routes with small category scores could be distant from the start point. Although the nearest neighbor-based approach finds the closest PoIs, it cannot efficiently find such PoIs. On the other hand, the Dijkstra-based approach searches for all PoI vertices that match a PoI category. Therefore, the Dijkstra-based approach is more suitable for the SkySR query than the nearest neighbor-based approach.

Although our approach finds sequenced routes simultaneously, it entails a large number of executions of the Dijkstra algorithm. This is because, since the number of PoI candidates increases, a large number of possible routes increases. The search space does not become small effectively. To effectively reduce the search space, we exploit the branch-and-bound algorithm, which uses the upper and lower bounds of a branch of the search space to solve an optimization problem effectively. With BSSR, each branch corresponds to each route. For the upper and lower bounds, we compute the bounds during finding the set of SkySRs. Specifically, we compute the upper bound of a route from the already found sequenced routes, and we compute the lower bound from the current searched route (i.e., not a sequenced route yet). With the upper and lower bounds, we can safely prune unnecessary routes to improve efficiency.

To further increase efficiency, we propose optimization techniques for BSSR. In order to exploit the branch-and-bound algorithm, it is necessary to initialize the upper bound. Thus, we first search for a sequenced route to initialize the upper bound. However, it may take high computational cost to find a sequenced route. Therefore, we propose a *nearest neighbor-based initial search method* (NNinit) that finds sequenced routes efficiently by greedily finding PoI vertices. In addition, to effectively update the upper bound, we assign a priority to each route and use the priority queue to efficiently find routes that are likely to give an effective upper bound. To compute the lower bound, we compute the possible minimum distance and add it to the length score of a route to safely prune unnecessary routes. Moreover, to avoid executing the Dijkstra algorithm iteratively from the same vertices, we materialize search results of the Dijkstra algorithm and reuse them to search the PoI vertices. By using BSSR with optimization techniques, we can perform the SkySR query efficiently.

5.2 Bulk SkySR algorithm

Bulk SkySR algorithm (BSSR) finds all SkySRs by finding simultaneously sequenced routes with checking dominance on demand. The naive solution must execute OSR queries for all super-category sequences of $\mathbf{S}_q$ one by one because it only searches for the PoIs that perfectly match the given category. In contrast, BSSR searches for all PoIs that semantically match the given category.

The basic process of BSSR is simple as shown in Algorithm 1: (i) start searching the PoI vertices that match the first category from start point v_q and insert the route found into priority queue Q_b which stores all found routes (line 4), (ii) fetch a route from Q_b (line 6), (iii) search for the next PoI vertices that semantically match the next category c_d from PoI vertex p_d which is the end of the fetched route, and insert the fetched route with each of the found PoI vertices into Q_b (lines 7–9), and (iv) if Q_b is not empty, return to (ii), otherwise output the minimal set of sequenced route $\mathcal{S}$ (line 10). In steps (i) and (iii), we find PoI vertices from the end of the fetched route using a Dijkstra algorithm modified for the SkySR query as described in Section 5.2.2.

Algorithm 1: Bulk SkySR algorithm

```
1  procedure BSSR(v_q, S_q)
2  S ← φ;
3  priority_queue Q_b ← φ;
4  mDijkstra(φ, c_S[1], v_q, Q_b, S);
5  while Q_b is not empty do
6      R ← Q_b.dequeue();
7      c_d ← c_S[|R| + 1];
8      p_d ← p_R[|R|];
9      mDijkstra(R, c_d, p_d, Q_b, S);
10 return S;
11 end procedure
```

5.2.1 Branch-and-bound. We search for sequenced routes simultaneously to reduce the search space. Our idea to safely reduce the search space is to exploit the branch-and-bound algorithm, which can reduce unnecessary search space. This section

describes the theoretical background of using the branch-and-bound algorithm. We use the following three lemmas to reduce the search space:

LEMMA 5.1. *Let $\mathcal{S}$ be a minimum set of sequenced routes while searching for SkySRs and $\mathcal{S}'$ be the minimum set of sequenced routes after finding SkySRs. If sequenced route* $\mathbf{R}$ *is dominated by a sequenced route in $\mathcal{S}$,* $\mathbf{R}$ *cannot be included in $\mathcal{S}'$.*

proof: From Definition 4.2, we search for a set of SkySRs, which are not dominated by the other sequenced routes. If we find a sequenced route not dominated by any sequenced routes in $\mathcal{S}$, we update $\mathcal{S}$ by inserting the new sequenced route and deleting a sequenced route dominated by the new one. Therefore, any sequenced routes in $\mathcal{S}$ after the update are not dominated by any sequenced routes in $\mathcal{S}$ prior to the update. As a result, sequenced routes in $\mathcal{S}'$ are not dominated by any sequenced routes in $\mathcal{S}$. In other words, $\mathbf{R}$ is not included in $\mathcal{S}'$ if we have sequenced route $\mathbf{R}'$ in $\mathcal{S}$ such that $l(\mathbf{R}') \leq l(\mathbf{R})$ and $s(\mathbf{R}') \leq s(\mathbf{R})$. □

LEMMA 5.2. *Let $\mathcal{E}(\mathbf{R})$ be a set of super-routes of* $\mathbf{R}$ *starting from the same start point. For any route $\mathbf{R}'$ in $\mathcal{E}(\mathbf{R})$, the length and semantic scores $l(\mathbf{R}')$ and $s(\mathbf{R}')$ cannot be less than $l(\mathbf{R})$ and $\underline{s}(\mathbf{R})$, respectively.*

proof: Let $\mathbf{R}'$ be a route included in $\mathcal{E}(\mathbf{R})$. Since we have $D(u_i, u_j) \geq 0$, the following property holds for a route $\mathbf{R}$ from Equation (1) of Definition 3.5.

$$\begin{aligned} & D(v_q, p_{R'}[1]) + \Sigma_{i=1}^{|\mathbf{R}'|-1} D(p_{R'}[i], p_{R'}[i+1]) \\ = \; & D(v_q, p_R[1]) + \Sigma_{i=1}^{|\mathbf{R}|-1} D(p_R[i], p_R[i+1]) \\ & \qquad + \Sigma_{i=|\mathbf{R}|}^{|\mathbf{R}'|-1} D(p_{R'}[i], p_{R'}[i+1]) \\ \geq \; & D(v_q, p_R[1]) + \Sigma_{i=1}^{|\mathbf{R}|-1} D(p_R[i], p_R[i+1]). \end{aligned}$$

Therefore, we have $l(\mathbf{R}) \leq l(\mathbf{R}')$. $\underline{s}(\mathbf{R})$ is the possible minimum semantic score of $\mathbf{R}$ when it becomes a sequenced route. Thus, even if PoI vertices are added to $\mathbf{R}$, we have $\underline{s}(\mathbf{R}) \leq s(\mathbf{R}')$. As a result, we have $l(\mathbf{R}) \leq l(\mathbf{R}')$ and $\underline{s}(\mathbf{R}) \leq s(\mathbf{R}')$. □

In terms of the branch-and-bound algorithm, Lemma 5.1 and 5.2 give us the upper and lower bounds of the scores of a route, respectively. We can prune routes according to the following lemma.

LEMMA 5.3. *(***pruning condition***) If (i)* $\mathbf{R}$ *is a sequenced route included in the set $\mathcal{S}$ of sequenced routes and (ii) $l(\mathbf{R}) \leq l(\mathbf{R}')$ and $s(\mathbf{R}) \leq \underline{s}(\mathbf{R}')$, any routes in $\mathcal{E}(\mathbf{R}')$ cannot be included in $\mathcal{S}$.*

proof: If we have $l(\mathbf{R}) \leq l(\mathbf{R}')$ and $s(\mathbf{R}) \leq \underline{s}(\mathbf{R}')$, $\mathbf{R}'$ is not included in $\mathcal{S}$ (Lemma 5.1). From Lemma 5.2, the scores of $\mathbf{R}'$ cannot become less than $l(\mathbf{R}')$ and $\underline{s}(\mathbf{R}')$ even if we expand $\mathbf{R}'$. Therefore, any routes in $\mathcal{E}(\mathbf{R}')$ cannot be included in $\mathcal{S}$ because $\mathbf{R}'$ is dominated by or equivalent to the sequenced route with $l(\mathbf{R})$ and $s(\mathbf{R})$. □

Lemma 5.3 gives us the length score *threshold* for a route, and, if the length score of a route is greater than this threshold, we can prune the given route. We define the length score threshold of a route as follows:

Definition 5.4. The threshold $\bar{l}(\mathbf{R})$ of the length score of route $\mathbf{R}$ is given by the following equation:

$$\bar{l}(\mathbf{R}) = \min_{\mathbf{R}' \in \mathcal{S}} \{ l(\mathbf{R}') | \underline{s}(\mathbf{R}) \geq s(\mathbf{R}') \}. \tag{3}$$

If $\bar{l}(\mathbf{R}) \leq l(\mathbf{R})$, we can safely prune $\mathbf{R}$ because it cannot be included in the result. Thus, we can reduce the search space without sacrificing the exactness of the result. Equation (3) has a small computation cost because $\mathcal{S}$ includes only a small number of sequenced routes as shown in Section 7.

5.2.2 The modified Dijkstra Algorithm. We search the next PoI vertices that semantically match the next PoI category using the modified Dijkstra algorithm. The modified Dijkstra algorithm can prune unnecessary routes based on Lemma 5.3. Moreover, based on the following lemma, it terminates unnecessary traversal of the graph and avoids inserting unnecessary routes.

LEMMA 5.5. *Let* $\mathbf{R} = \langle p_R[1], \ldots, p_R[i], p_R[i+1], p_R[i+2] \ldots, p_R[|\mathbf{R}|] \rangle$ *be a route and $p_{i:i+1}$ be a PoI vertex on a path between $p_R[i]$ and $p_R[i+1]$. Route* $\mathbf{R}$ *must be dominated by or equivalent to another route if we have $sim(c_S[i+1], c_{p_{i:i+1}}) \geq sim(c_S[i+1], c_{p_R[i+1]})$.*

proof: Let $\mathbf{R}' = \langle p_R[1], \ldots, p_R[i], p_{i:i+1}, p_R[i+2], \ldots, p_R[|\mathbf{R}|] \rangle$ be a route such that the difference between $\mathbf{R}$ and $\mathbf{R}'$ is only in $p_{i:i+1}$ and $p_R[i+1]$. Since the PoI vertex $p_{i:i+1}$ is on the path between $p_R[i]$ and $p_R[i+1]$, we have $l(\mathbf{R}) \geq l(\mathbf{R}')$ based on triangle inequality (i.e., $D(p_{i:i+1}, p_R[i+1]) + D(p_R[i+1], p_R[i+2]) \geq D(p_{i:i+1}, p_R[i+2])$). Moreover, if $sim(c_S[i+1], c_{p_{i:i+1}}) \geq sim(c_S[i+1], c_{p_R[i+1]})$, we have $s(\mathbf{R}) \geq s(\mathbf{R}')$. Therefore, $\mathbf{R}$ is dominated by or equivalent to $\mathbf{R}'$ because $l(\mathbf{R}) \geq l(\mathbf{R}')$ and $s(\mathbf{R}) \geq s(\mathbf{R}')$. □

Lemma 5.5 gives us two properties for the SkySR query: (i) even if we find a PoI vertex that passes through another PoI vertex that has a better category similarity, we can ignore the PoI vertex, and (ii) if we find a PoI vertex that perfectly matches the given category, we do not need to traverse the graph through the PoI vertex. As a result, using Lemma 5.3 and 5.5, we can efficiently find the next PoI vertices.

Algorithm 2 shows the pseudocode for the modified Dijkstra algorithm, which is used to find PoI vertices that semantically match c_d from p_d. In priority queue Q_d for the modified Dijkstra algorithm, the top vertex is the closest vertex to p_d. The queue is initialized to p_d (line 3). The closest vertex to p_d is dequeued from Q_d (line 5). $\mathbf{R}_t$ is a route expanded from $\mathbf{R}_d$, which is $\mathbf{R}_d$ with fetched vertex u (line 7). If the length score of $\mathbf{R}_t$ is greater than or equal to the threshold of $\mathbf{R}_d$, the modified Dijkstra algorithm terminates the process (Lemma 5.3) (line 8). We check whether (i) u semantically matches c_d and (ii) u does not proceed through another PoI vertex whose category similarity is greater than or equal to that of u (line 9). If we satisfy the above conditions and the length score of $\mathbf{R}_t$ is less than its threshold (line 10), we insert $\mathbf{R}_t$ into the priority queue or the set of sequenced routes (lines 10–12). Otherwise, we skip the process to insert $\mathbf{R}_t$ (Lemma 5.3 and 5.5). The neighbor vertices of u are inserted into Q_d unless u perfectly matches c_d (Lemma 5.5) (lines 13–17).

5.3 Optimization techniques

In this section, we propose four optimization techniques for BSSR. Section 5.3.1 explains an initial search for sequenced routes and proposes NNinit. We then explain tightening the upper and the lower bounds in Section 5.3.2 and Section 5.3.3, respectively. Furthermore, in Section 5.3.4 we propose an *on-the-fly caching technique* to reuse previous search results of the modified Dijkstra algorithm.

5.3.1 Initial search. We prune unnecessary routes efficiently using the branch-and-bound algorithm. However, we cannot calculate the threshold of $\mathbf{R}$ if there are no sequenced routes in $\mathcal{S}$ whose semantic scores are not greater than that of $\underline{s}(\mathbf{R})$ based

Algorithm 2: Modified Dijkstra algorithm to find the next PoI vertices matching c_d from p_d

```
1  procedure mDijkstra(R_d, c_d, p_d, Q_b, S)
2  dist[u] = inf for all u ∈ V ∪ P, dist[p_d] = 0;
3  priority_queue Q_d ← {p_d};
4  while Q_d is not empty do
5      u ← Q_d.dequeue;
6      if u is already visited then continue;
7      R_t ← R_d ⊕ u;
8      if l(R_t) ≥ l̄(R_d) then break;
9      if u ∈ P_{t_{c_d}} and u is not through the PoI vertex whose category
       similarity is higher than that of u then
10         if l(R_t) < l̄(R_t) then
11             if R_t is a sequenced route then S.update(R_t);
12             else Q_b.enqueue(R_t);
13     if u ∉ P_{c_d} then
14         for each u' for e(u, u') ∈ E do
15             if dist[u] + w(u, u') < dist[u] then
16                 dist[u'] = dist[u] + w(u, u').w;
17                 Q_d.enqueue(u');
18 end procedure
```

Algorithm 3: Initial search for finding sequenced routes with a small cost

```
1  procedure NNinit(v_q, S_q)
2  S ← φ, R ← φ;
3  priority_queue Q ← {v_q};
   /* execute Dijkstra algorithm |S_q| times                          */
4  for i : 1 to |S_q| do
5      dist[u] = inf for all u ∈ V ∪ P, dist[Q.top] = 0;
6      while Q is not empty do
7          u ← Q.dequeue;
8          if u is already visited then continue;
9          if i = |S_q| and u ∈ P_{t_{c_{S_q}[i]}} then
10             R' ← R ⊕ u;
11             S.update(R');
12         if u ∈ P_{c_{S_q}[i]} then
13             R ← R ⊕ u;
14             Q ← {u};
15             break;
16         for each u' for e(u, u') ∈ E do
17             if dist[u] + w(u, u') < dist[u'] then
18                 dist[u'] = dist[u] + w(u, u');
19                 Q.enqueue(u');
20 return S;
21 end procedure
```

on Equation (3). Therefore, initially, we search for the sequenced route whose semantic score is 0. However, the length score of the sequenced route can be large if its semantic score is 0. To tighten the threshold, we also search for sequenced routes whose semantic scores are greater than 0 because the length scores of them are less than that of the sequenced route with a semantic score of 0. We initially find several sequenced routes to tighten the upper bound.

We propose NNinit, which searches for several sequenced routes efficiently. NNinit performs a nearest neighbor search repeatedly to find PoI vertices that perfectly match the given categories. With this process, we can find a sequenced route whose semantic score is 0. Moreover, NNinit can find the PoI vertex that semantically matches the given category during the nearest neighbor search. When we find the last visited PoI vertex, we may find PoI vertices that semantically match the last category in S_q. Therefore, we can obtain sequenced routes whose semantic scores are greater than 0 and length scores are small. As a result, NNinit can find several sequenced routes without incurring additional cost, and one of the sequenced routes has a semantic score of 0.

We present the pseudocode for NNinit in Algorithm 3. Here, priority queue Q is initialized to start point v_q (line 3). NNinit repeats the Dijkstra algorithm $|\mathrm{S}_q|$ times to find sequenced routes (line 4). The Dijkstra algorithm is executed to search for the closest PoI vertex that perfectly matches $c_{S_q}[i]$ from the initial vertex (the first initial vertex is v_q) (lines 5–19). Here, the closest vertex to the initial vertex is dequeued from Q (line 7). If the algorithm finds a PoI vertex that perfectly matches $c_{S_q}[i]$, this vertex is added to $\mathbf{R}$ and Q is initialized to the PoI vertex (lines 12–15). When it finds the last PoI vertex that semantically matches $c_{S_q}[|\mathrm{S}_q|]$, it inserts the sequenced route into $\mathcal{S}$ (lines 9–11). Finally, we obtain a set of sequenced routes, and one of the sequenced routes in $\mathcal{S}$ has a semantic score of 0.

Example 5.6. We show an example of NNinit using Example 1.1, which searches an Asian restaurant, an A&E place, and a gift shop in this order from start point v_q. NNinit executes the Dijkstra algorithm three times because the size of category sequence is three. First, NNinit searches PoI vertices that perfectly match Asian restaurant from v_q. Then, it finds p_2 that is the closest PoI that perfectly match Asian restaurant to v_q. Next, it searches the closest PoI vertex that perfectly matches A&E to p_2 and then finds p_5. From the next search, NNinit inserts sequenced routes to $\mathcal{S}$ when it finds PoI vertices that semantically match gift shop. NNinit finds p_7 whose category is Shop&Service (i.e., semantically match) and thus inserts $\langle p_2, p_5, p_7\rangle$ to $\mathcal{S}$. After finding p_7, it finds p_8 that perfectly matches gift shop and inserts $\langle p_2, p_5, p_8\rangle$ to $\mathcal{S}$. Finally NNinit returns $\mathcal{S}$ including $\{\langle p_2, p_5, p_8\rangle, \langle p_2, p_5, p_7\rangle\}$. The length score of $\langle p_2, p_5, p_7\rangle$ is 12, which is less than the length score of $\langle p_2, p_5, p_8\rangle$ of 15.

5.3.2 Tightening upper bound: Arranging routes in the priority queue. We use the upper bound to prune unnecessary routes. The upper bound is computed from the obtained sequenced routes. To tighten the upper bound, it is important to efficiently find sequenced routes that have small length and semantic scores. BSSR extends a route at the top of the priority queue to search for a sequenced route, as shown in Algorithm 1. Note that priority queues in existing algorithms conventionally consider only distances (i.e., a distance-based priority queue). If we use a distance-based priority queue, BSSR preferentially extends a route with a small length score. Although we must increase the size of a route to $|\mathrm{S}_q|$ to find a sequenced route, a route that has a small length score likely has a small size. Therefore, it is difficult to search for sequenced routes efficiently using a distance-based priority queue.

To search for sequenced routes efficiently, we preferentially extend a route that has a large size. Here, since many routes in the priority queue could have the same size, we must consider an additional priority, which is expected to affect performance. If multiple routes in the priority queue are the same size, we preferentially extend the route with the smallest semantic score. We can reduce the search space by searching for sequenced routes in ascending order of semantic score. Moreover, if routes are the same size and have the same semantic score, we preferentially extend the route with the smallest length score. As a result, we can efficiently obtain sequenced routes with small length and semantic scores.

5.3.3 Tightening lower bound: Possible minimum length score. As described in Section 5.2.1, we use the length scores of routes as the lower bound, i.e., we prune a route if the length score of the route is not less than the threshold. Note that the length score of the route increases as the route size increases. This indicates that it is difficult to prune routes before the route size increases. Our approach to tighten the lower bound of the route is to estimate the increase of the length score. However, if we carelessly estimate a future length score, we may sacrifice the exactness of th result.

The basic idea of this estimation is to calculate the *possible minimum distance*. Here, we compute the smallest distance among any pair of PoI vertices in sets of PoI vertices. We use the following two minimum distances, *semantic-match minimum distance* $\underline{l_s}$ and *perfect-match minimum distance* $\underline{l_p}$:

Definition 5.7. (**minimum distance**) The semantic-match minimum distance $\underline{l_s}$ and perfect-match minimum distance $\underline{l_p}$ are given by the following equations:

$$\underline{l_s}(\mathbf{R}) = \Sigma_{i=|\mathbf{R}|}^{|\mathbf{S}_q|-1} \underline{l_s}[i], \text{where } \underline{l_s}[i] = \min_{p_i \in \mathbb{P}_{t_i}, p_{i+1} \in \mathbb{P}_{t_{i+1}}} D(p_i, p_{i+1}). \quad (4)$$

$$\underline{l_p}(\mathbf{R}) = \Sigma_{i=|\mathbf{R}|}^{|\mathbf{S}_q|-1} \underline{l_p}[i], \text{where } \underline{l_p}[i] = \min_{p_i \in \mathbb{P}_{t_i}, p_{i+1} \in \mathbb{P}_{c_{i+1}}} D(p_i, p_{i+1}). \quad (5)$$

In Equations (4) and (5), $\mathbb{P}_{t_i}$ and $\mathbb{P}_{c_i}$ denote the set of PoI vertices associated with a category tree of $c_{S_q}[i]$ and the set of PoI vertices whose category is $c_{S_q}[i]$, respectively.

We compute the semantic-match minimum distance based on the distance to the PoI vertices that semantically match the next category. We can safely add the semantic-match minimum distance to the current length score without restriction. However, the semantic-match minimum distance is much less than the threshold. Thus, it could be difficult to improve pruning performance; thus, we use the perfect-match minimum distance to increase pruning performance. The perfect-match minimum distance is computed based on the distance to the PoI vertices that perfectly match the next category. We can improve pruning performance using the perfect-match minimum distance compared to the semantic-match minimum distance because the perfect-match minimum distance is much greater than the semantic-match minimum distance; therefore, the perfect-match minimum distance tightens the lower bound more than the semantic-match minimum distance. However, we can use the perfect-match minimum distance only in a special case, i.e., where a route must pass only PoIs that perfectly match the given categories so as not to be dominated. The perfect-match minimum distance works well if the number of sequenced route in $\mathcal{S}$ is large because the constraint is usually satisfied by increasing the number of sequenced route in $\mathcal{S}$.

Lemma 5.8. *Let* $\mathbf{R}'$ *and* $\mathbf{R}''$ *be sequenced routes in* $\mathcal{S}$ *and* $\mathbf{R}$ *be a route such that (i)* $l(\mathbf{R}) \geq l(\mathbf{R}')$ *and* $s(\mathbf{R}) < s(\mathbf{R}')$ *and (ii)* $l(\mathbf{R}) < l(\mathbf{R}'')$ *and* $s(\mathbf{R}) \geq s(\mathbf{R}'')$. *Let* δ *be the minimum increment of a semantic score*[2]. *We can prune* $\mathbf{R}$ *if we have (a)* $l(\mathbf{R}) \geq l(\mathbf{R}')$ *and* $s(\mathbf{R}) + \delta \geq s(\mathbf{R}')$ *and (b)* $l(\mathbf{R}) + \underline{l_p}(\mathbf{R}) \geq l(\mathbf{R}'')$ *and* $s(\mathbf{R}) \geq s(\mathbf{R}'')$.

proof: First, we consider case (a). If we have $l(\mathbf{R}) \geq l(\mathbf{R}')$ and $s(\mathbf{R}) + \delta \geq s(\mathbf{R}')$, $\mathbf{R}$ is dominated by or equivalent to $\mathbf{R}'$ if its semantic score increases. Therefore, $\mathbf{R}$ must only pass through PoI vertices that perfectly match the given categories not to be dominated. If $\mathbf{R}$ passes through only PoI vertices that perfectly match the given categories, the length score of $\mathbf{R}$ increases by at least $\underline{l_p}(\mathbf{R})$. For case (b), if we have $l(\mathbf{R}) + \underline{l_p}(\mathbf{R}) \geq l(\mathbf{R}'')$ and $s(\mathbf{R}) \geq s(\mathbf{R}'')$, $\mathbf{R}$ is dominated by or equivalent to $\mathbf{R}''$ if its length score increases by $\underline{l_p}(\mathbf{R})$. As a result, if we have two routes $\mathbf{R}'$ and $\mathbf{R}''$, such as (i) $l(\mathbf{R}) \geq l(\mathbf{R}')$ and $s(\mathbf{R}) + \delta \geq s(\mathbf{R}')$ and (ii) $l(\mathbf{R}) + \underline{l_p}(\mathbf{R}) \geq l(\mathbf{R}'')$ and $s(\mathbf{R}) \geq s(\mathbf{R}'')$, $\mathbf{R}$ is dominated by or equivalent to at least one of $\mathbf{R}'$ and $\mathbf{R}''$. □

To compute the estimation of the lower bound, we compute two types of possible minimum distances $\underline{l_s}$ and $\underline{l_p}$. A naive approach computes all minimum distances from the PoI vertices that semantically match $c_{S_q}[i]$ to $c_{S_q}[i+1]$ for $1 \leq i \leq |\mathbf{S}_q| - 1$ by iteratively executing the Dijkstra algorithm. However, this has a high computational cost. To reduce the cost, we execute a *multi-source multi-destination Dijkstra algorithm*. In this algorithm, all start points are inserted into the same priority queue. Then, the algorithm dequeues vertices in the same manner as the conventional Dijkstra algorithm. Here, the process is terminated if the top of the priority queue becomes one of the destinations. This approach only needs $|\mathbf{S}_q| - 1$ times to compute the possible minimum distance. The multi-source multi-destination Dijkstra algorithm guarantees the minimum distance by the following lemma:

Lemma 5.9. *The multi-source multi-destination Dijkstra algorithm guarantees the minimum distance from the start points to the destinations.*

proof: We first insert multiple start points into the priority queue, and their distances from the start points are initialized as 0. If we find a vertex, it is inserted into the queue and the distance to the vertex is updated from the closest start point to the vertex. The vertex with the smallest distance from the start point in the priority queue is dequeued from the priority queue. If the top vertex in the priority queue is one of the destinations, there are no destinations with smaller distance than the top one. Therefore, we can guarantee the minimum distance from the start points to the destinations. □

Algorithm 4 shows the pseudocode to compute the semantic-match minimum distance. The estimation of the lower bound is executed after line 4 in Algorithm 1. Here, we initialize $\mathbb{P}_i$ and $\mathbb{P}_{i+1}$ (lines 3–4). $\bar{l}(\phi)$ denotes the threshold for a route whose semantic score is 0. The difference between computing the semantic-match and perfect-match minimum distances is whether the PoI vertices in $\mathbb{P}_{i+1}$ semantically or perfectly match the given category.

Example 5.10. We show an example to compute the semantic-match minimum distance using Example 1.1. $\mathbb{P}_1$, $\mathbb{P}_2$, and $\mathbb{P}_3$ include $\{p_1, p_2, p_6, p_{10}, p_{11}\}$, $\{p_5, p_9, p_{12}\}$, and $\{p_3, p_4, p_7, p_8, p_{13}\}$, respectively. First, PoI vertices in $\mathbb{P}_1$ are inserted to priority queue Q, and the set of destinations is $\mathbb{P}_2$. By processing the Dijkstra algorithm, we compute possible minimum distance $\underline{l_s}[1] = 2$ (from p_6 to p_9). Next, we search PoI vertices that semantically match A&E to gift shop. Then, we compute $\underline{l_s}[2] = 1$ (from p_{12} to p_{13}). Finally, we obtain semantic-match minimum distance $\underline{l_s} = \{2, 1\}$. We can compute the perfect-match minimum distance in the same way and obtain $\underline{l_p} = \{3, 1\}$, which is greater than $\underline{l_s}$.

5.3.4 Reuse of the temporal result: On-the-fly caching technique. Although BSSR efficiently prunes unnecessary routes, it may iteratively execute the modified Dijkstra algorithm at the same vertex because, in Algorithm 1 (line 8), p_d could be the

[2]The least increase of the semantic score is computed from the category tree. Specifically, we can compute the least increase from the category that is most similar (but not equal) to the next category.

Algorithm 4: Computing possible minimum distance

```
1  procedure EstimationLowerbound(v_q, S_q)
2  for i : 1 to |S_q| − 1 do
3      ℙ_i ← {p | p ∈ ℙ_{t_{c_{S_q}[i]}} and D(v_q, p) < l̄(φ)};
4      ℙ_{i+1} ← {p | p ∈ ℙ_{t_{c_{S_q}[i+1]}} and D(v_q, p) < l̄(φ)};
5      dist[u] = inf for all u ∈ 𝕍 ∪ ℙ, dist[p] = 0 for all p ∈ ℙ_i;
6      priority_queue Q ← {p} ∈ ℙ_i;
7      while Q is not empty do
8          u ← Q.dequeue;
9          if u is already visited then continue;
10         if u ∈ ℙ_{i+1} then
11             l_s[i] = dist[u];
12             break;
13         for each u′ for e(u, u′) ∈ E do
14             if dist[u] + w(u, u′) < dist[u′] then
15                 dist[u′] = dist[u] + w(u, u′);
16                 Q.enqueue(u′);
17 return l_s;
18 end procedure
```

same as the former executions of the modified Dijkstra algorithms. Thus, we reuse the result starting at the same PoI vertex by materializing the result of the modified Dijkstra algorithm (i.e., keeping PoI vertices matching c_d and distances from p_d to the PoI vertices), which we refer to as *on-the-fly caching*.

After finding SkySRs, on-the-fly caching frees the results of the modified Dijkstra algorithms (this is why we call it *on-the-fly*), because the search space rarely overlaps across different inputs (i.e., S_q and v_q differ).

5.4 Theoretical Analysis

In this section, we theoretically analyze the cost and correctness of the proposed BSSR.

Theorem 1. **(Time complexity)** *Let γ be a ratio of pruning and α be a ratio of the size of a graph to find the SkySRs. The time complexity of* BSSR *is* $O(\gamma(\alpha|\mathbb{P}|)^{|\mathrm{S}_q|}\alpha(|\mathbb{E}|+(|\mathbb{V}|+|\mathbb{P}|)\log(\alpha(|\mathbb{V}|+|\mathbb{P}|)))))$.

proof: The time complexity of the Dijkstra algorithm is $O(|\mathbb{E}|+|\mathbb{V}|\log|\mathbb{V}|)$ if the number of vertices is $|\mathbb{V}|$. In our setting, we have $|\mathbb{V}|+|\mathbb{P}|$ vertices because we have two types of vertices. In addition, we do not need to search the whole graph by reducing the graph size according to the threshold. Therefore, the time complexity of the modified Dijkstra algorithm is $O(\alpha(|\mathbb{E}|+(|\mathbb{V}|+|\mathbb{P}|)\log(\alpha(|\mathbb{V}|+|\mathbb{P}|))))$. The time complexity of BSSR depends on the number of times the modified Dijkstra algorithms is executed. The number of modified Dijkstra algorithms is equal to all the potential routes $|\mathbb{P}|^{|\mathrm{S}_q|}$. Recall that we can prune the number of routes using the branch-and-bound algorithm. Finally, the time complexity of BSSR is $O(\gamma(\alpha|\mathbb{P}|)^{|\mathrm{S}_q|}\alpha(|\mathbb{E}|+(|\mathbb{V}|+|\mathbb{P}|)\log(\alpha(|\mathbb{V}|+|\mathbb{P}|)))))$. □

In our approach, γ and α depend on the upper and lower bounds. These are affected by the graph structure, the category trees, and the ratio of PoI vertices, and the time complexity of BSSR depends on these factors.

Theorem 2. **(Space complexity)** *Let γ be the pruning ratio, and α be the ratio of the size of the graph to find the SkySRs. The space complexity of* BSSR *is* $O(|\mathbb{E}|+|\mathbb{V}|+|\mathbb{P}|+\gamma|\mathrm{S}_q|(\alpha|\mathbb{P}|)^{|\mathrm{S}_q|})$.

proof: We store the whole graph of size $O(|\mathbb{E}|+|\mathbb{V}|+|\mathbb{P}|)$. We also store routes into the priority queue and $\mathcal{S}$, and the maximum number of routes is $|\mathbb{P}|^{|\mathrm{S}_q|}$. We can prune the number of routes using the branch-and-bound algorithm. The size of the routes is proportional to $|\mathrm{S}_q|$. Therefore, the space complexity of BSSR is $O(|\mathbb{E}|+|\mathbb{V}|+|\mathbb{P}|+\gamma|\mathrm{S}_q|(\alpha|\mathbb{P}|)^{|\mathrm{S}_q|})$. □

If the number of routes in the priority queue is small, the graph size becomes the main factor related to the memory usage. Otherwise, the number of routes in the priority queue is the main factor.

Theorem 3. **(Correctness)** BSSR *guarantees the exact result.*

proof: BSSR prunes routes based on the upper and lower bounds. BSSR safely prunes routes dominated by or equivalent to the obtained sequenced routes. As a result, BSSR does not sacrifice the exactness of the search result. □

5.5 Running Example

We demonstrate BSSR with optimization techniques using Example 1.1. Table 4 shows routes in priority queue Q_b and sequenced routes in $\mathcal{S}$. To compute category similarity and semantic score, we use Equations (6) and (7), respectively.

First, we process NNinit, and $\mathcal{S}$ initially includes $\{\langle p_2, p_5, p_8\rangle, \langle p_2, p_5, p_7\rangle\}$. 1st step: BSSR starts to find PoI vertices that semantically match Asian restaurant from v_q with the threshold of 15. Then, it finds p_1, p_2, p_6, p_{10}, and p_{11}. Both p_2's and p_{10}'s category similarities are 1, and their lengths are 6 and 8, respectively. Thus, p_2 comes the top in Q_b. 2nd step: BSSR searches PoI vertices that semantically match Arts&Entertainment from p_2, and finds p_5. Since $\langle p_2, p_{12}\rangle$ passes through p_5 and $l(\langle p_2, p_9\rangle)$ is more than 15, both routes are not inserted to Q_b. 3rd step: as the top route is $\langle p_2, p_5\rangle$, BSSR searches PoI vertices that semantically match gift shop from p_5. BSSR does not find any routes due to the threshold. 4th step: BSSR fetches $\langle p_{10}\rangle$ from Q_b and inserts two routes $\langle p_{10}, p_5\rangle$ and $\langle p_{10}, p_{12}\rangle$ to Q_b. 5th step: BSSR fetches $\langle p_{10}, p_{12}\rangle$ and finds sequenced route $\langle p_{10}, p_{12}, p_{13}\rangle$. Since $\langle p_{10}, p_{12}, p_{13}\rangle$ dominates $\langle p_2, p_5, p_8\rangle$, $\langle p_2, p_5, p_8\rangle$ is deleted from $\mathcal{S}$. 6th step: The top route $\langle p_{10}, p_5\rangle$ is deleted from Q_b because its length score is not smaller than the threshold of 13. 7th step: BSSR fetches $\langle p_1\rangle$ and inserts $\langle p_1, p_5\rangle$ and $\langle p_1, p_9\rangle$. 8th step: BSSR fetches $\langle p_1, p_9\rangle$ and finds a sequenced route $\langle p_1, p_9, p_8\rangle$. $\langle p_1, p_9, p_8\rangle$ is inserted to $\mathcal{S}$, and $\langle p_2, p_5, p_7\rangle$ is deleted from $\mathcal{S}$. 9th step: $\langle p_1, p_5\rangle$ is deleted due to the threshold. 10th step: BSSR fetches $\langle p_6\rangle$ and finds a route $\langle p_6, p_9\rangle$. 11th step: BSSR finds a sequenced route $\langle p_6, p_9, p_8\rangle$, and the route dominates $\langle p_1, p_9, p_8\rangle$. 12th step: The distance from p_{11} to the PoI vertices that match A&E is larger than the threshold. Finally, BSSR returns the set of SkySRs $\mathcal{S}$.

6 VARIATIONS AND EXTENSIONS

The SkySR query has a number of variations and extensions. We discuss some of these in the following.

Directed graphs: The SkySR query can be easily applied to directed graphs. We only need to use the Dijkstra algorithm for directed graphs. Here, no modification of the main idea is required.

PoI with multiple categories: To treat PoIs with multiple categories, we can change the definitions of sequenced routes and category similarity. Specifically, we change condition (ii) in Definition 3.4 to state that at least one $c_{p_i}[j]$ $(1 \le j \le k_i)$ semantically matches $c_S[i]$ for $1 \le i \le |\mathrm{S}|$, where $c_{p_i}[j]$ is the j-th category of p_i and k_i is the number of categories associated with p_i. The category similarity is either the highest or the average value among the category similarities.

Table 4: Example of BSSR algorithm

0	Q_b: $\mathcal{S}$: $\langle p_2, p_5, p_8\rangle$, $\langle p_2, p_5, p_7\rangle$
1	Q_b: $\langle p_2\rangle$, $\langle p_{10}\rangle$, $\langle p_1\rangle$, $\langle p_6\rangle$, $\langle p_{11}\rangle$ $\mathcal{S}$: $\langle p_2, p_5, p_8\rangle$, $\langle p_2, p_5, p_7\rangle$
2	Q_b: $\langle p_2, p_5\rangle$, $\langle p_{10}\rangle$, $\langle p_1\rangle$, $\langle p_6\rangle$, $\langle p_{11}\rangle$ $\mathcal{S}$: $\langle p_2, p_5, p_8\rangle$, $\langle p_2, p_5, p_7\rangle$
3	Q_b: $\langle p_{10}\rangle$, $\langle p_1\rangle$, $\langle p_6\rangle$, $\langle p_{11}\rangle$ $\mathcal{S}$: $\langle p_2, p_5, p_8\rangle$, $\langle p_2, p_5, p_7\rangle$
4	Q_b: $\langle p_{10}, p_{12}\rangle$, $\langle p_{10}, p_5\rangle$, $\langle p_1\rangle$, $\langle p_6\rangle$, $\langle p_{11}\rangle$ $\mathcal{S}$: $\langle p_2, p_5, p_8\rangle$, $\langle p_2, p_5, p_7\rangle$
5	Q_b: $\langle p_{10}, p_5\rangle$, $\langle p_1\rangle$, $\langle p_6\rangle$, $\langle p_{11}\rangle$ $\mathcal{S}$: $\langle p_{10}, p_{12}, p_{13}\rangle$, $\langle p_2, p_5, p_7\rangle$
6	Q_b: $\langle p_1\rangle$, $\langle p_6\rangle$, $\langle p_{11}\rangle$ $\mathcal{S}$: $\langle p_{10}, p_{12}, p_{13}\rangle$, $\langle p_2, p_5, p_7\rangle$
7	Q_b: $\langle p_1, p_9\rangle$, $\langle p_1, p_5\rangle$, $\langle p_6\rangle$, $\langle p_{11}\rangle$ $\mathcal{S}$: $\langle p_{10}, p_{12}, p_{13}\rangle$, $\langle p_2, p_5, p_7\rangle$
8	Q_b: $\langle p_1, p_5\rangle$, $\langle p_6\rangle$, $\langle p_{11}\rangle$ $\mathcal{S}$: $\langle p_{10}, p_{12}, p_{13}\rangle$, $\langle p_1, p_9, p_8\rangle$
9	Q_b: $\langle p_6\rangle$, $\langle p_{11}\rangle$ $\mathcal{S}$: $\langle p_{10}, p_{12}, p_{13}\rangle$, $\langle p_1, p_9, p_8\rangle$
10	Q_b: $\langle p_6, p_9\rangle$, $\langle p_{11}\rangle$ $\mathcal{S}$: $\langle p_{10}, p_{12}, p_{13}\rangle$, $\langle p_1, p_9, p_8\rangle$
11	Q_b: $\langle p_{11}\rangle$ $\mathcal{S}$: $\langle p_{10}, p_{12}, p_{13}\rangle$, $\langle p_6, p_9, p_8\rangle$
12	Q_b: $\mathcal{S}$: $\langle p_{10}, p_{12}, p_{13}\rangle$, $\langle p_6, p_9, p_8\rangle$

Complex category requirement: We can specify more detailed category requirements, such as *conjunction*, *disjunction*, and *negation*. For example, we can specify that a PoI category is "American restaurant" or "Mexican restaurant" (disjunction), but not "Taco Place" (negation). If PoI vertices are associated with more than two categories, we can specify a conjunction such as "Cafe" and "Bakery". Note that the time complexity of our algorithm does not change if we specify a detailed requirement because the detailed requirements are equivalent to increasing the number of categories.

Skyline trip planning query: The proposed algorithm can be applied to the trip planning query without category order. For searching routes without category order, the proposed algorithm searches PoI vertices that semantically match a category in a given set of categories. Then, if the algorithm finds PoI vertices, it deletes the categories that are already included in the routes to find next PoI vertices. Note that we need to modify some definition and scoring functions for routes without category order. By this procedure, we can find skyline routes efficiently.

SkySR with destination: Note that we can specify the destination. The simple way to calculate a SkySR with a destination is to add the distance from the last visited PoI vertex to the destination to the length score after finding the sequenced route. To improve efficiency, we traverse PoI vertices from both the destination and the start point.

7 EXPERIMENTAL STUDY

We perform experiments to evaluate the effectiveness of the proposed algorithm. All algorithms are implemented in C++ and run on an Intel(R) Xeon(R) CPU E5620 @ 2.40GHz with 32 GB of RAM.

7.1 Experimental settings

Algorithm. We compare the proposed BSSR and algorithms that iteratively find OSRs using the Dijkstra-based solution and the PNE approach (denoted Dij and PNE, respectively), as described in Section 3. We evaluate performance with respect to (i) response time, and (ii) maximum resident set size (RSS) to represent memory usage.

Table 5: Summery of dataset

Dataset	Area	$\lvert\mathbb{V}\rvert$	$\lvert\mathbb{P}\rvert$	$\lvert\mathbb{E}\rvert$
Tokyo	Tokyo	401,893	174,421	499,397
NYC	New York city	1,150,744	451,051	1,722,350
Cal	California	21,048	87,365	108,863

Dataset. We conduct experiments using various maps (Tokyo, New York city, and California). Table 5 summarizes each dataset. For the Tokyo and NYC datasets, the road network is extracted from OpenStreetMap[3] and the PoI information is extracted from Foursquare. Each PoI is embedded on the closest edge in the same way as [10] and is associated with the Foursquare category trees. Note that the number of category trees in Foursquare is 10. For the Cal dataset, the road network and PoI information are available online[4]. The number of categories in the Cal dataset is 63[5]. For each dataset, we use distances based on longitude and latitude as edge weights and treat the graphs as undirected graphs. The graphs are implemented using adjacency lists.

For each dataset, we generate 100 searches, in which the size of a sequence is $|\mathrm{S}_q|$. The start points are selected randomly from vertices in the maps. The categories of sequences are selected randomly from the leaf nodes in the category trees with the constraint that they have different category trees. Since the number of PoI vertices associated with each category is significantly biased, we select only categories that have a large number of PoI vertices.

Here, category similarity is calculated based on the *Wu and Palmer similarity measure* [19] and the semantic score is calculated as the product of the category similarities of the sequence members. Specifically, we calculate the category similarity and semantic score using the following equations:

$$sim(c, c') = \max_{c_i \in a(c')} \frac{2 \cdot d(c_m)}{d(c) + d(c')}, \quad (6)$$

$$s(\mathbf{R}) = 1 - \Pi_{i=1}^{\min(|\mathbf{R}|, |\mathrm{S}_q|)} sim(c_{p_R[i]}, c_{S_q}[i]), \quad (7)$$

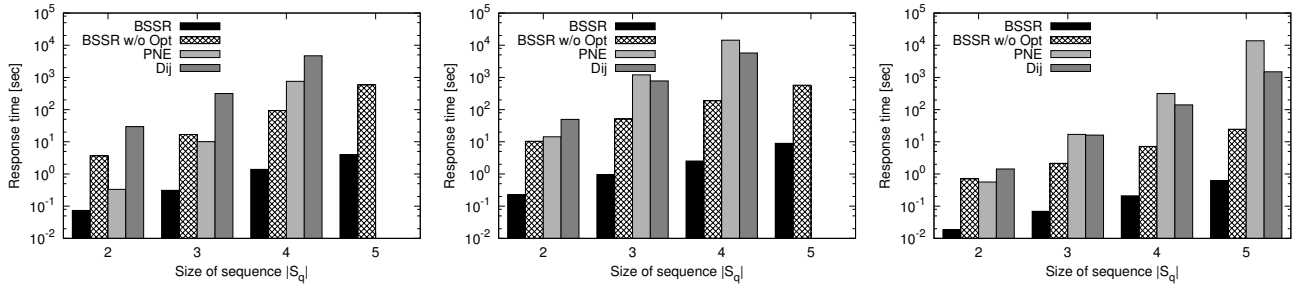
where $a(c)$, $d(c)$, and c_m denote the set of ancestor categories of c (including c), the depth of c, and the deepest common ancestor category of c and c_i, respectively.

7.2 Overview of results

First, we present an overview of the performance of all algorithms. Figure 3 shows the response time with various category sequence sizes, and Table 6 shows the RSS for a category sequence of size four. Here, "BSSR w/o Opt" denotes BSSR without optimization techniques. In Figure 3, there are missing bars for the case of size of sequence 5, because the executions were not finished after a month.

BSSR achieves the least response time with all datasets and reduces the search space by exploiting the branch-and-bound algorithm and the proposed optimization techniques. By comparing BSSR and BSSR w/o Opt, we confirm that the optimization techniques increase efficiency. When the size of the category sequence is small, PNE finds SkySRs efficiently because it can search for sequenced routes efficiently if the category sequence size is small. On the other hand, as category sequence size increases, the response time of PNE and Dij increases significantly.

[3] https://www.openstreetmap.org
[4] http://www.cs.utah.edu/~lifeifei/SpatialDataset.htm
[5] Since the PoIs in the Cal dataset have no category tree information, we generate a category of height three where a non-leaf node has three child nodes.



(a) Tokyo (b) NYC (c) Cal

Figure 3: Results obtained for the datasets with various $|S_q|$

Table 6: RSS Comparison

	BSSR	BSSR w/o Opt	PNE	Dij
Tokyo	239.6 MB	497.5 MB	239.8 MB	4.8 GB
NYC	658.0 MB	659.4 MB	658.7 MB	9.7 GB
Cal	36.7 MB	53.7 MB	36.6 MB	70.3 MB

Table 7: Effect of initial search for various $|S_q|$

Dataset	Approach	Metrics	2	3	4	5
Tokyo	Proposed	Weight sum	0.009	0.013	0.017	0.021
		Response time [msec]	3.5	5.1	6.9	8.6
		# of routes	1.49	1.33	1.36	1.49
		Ratio	0.74	0.79	0.82	0.86
	Existing	Weight sum	0.32 (regardless $\|S_q\|$)			
NYC	Proposed	Weight sum	0.044	0.066	0.073	0.078
		Response time [msec]	10.7	16.5	19.5	24.1
		# of routes	1.76	1.79	1.81	1.82
		Ratio	0.67	0.81	0.85	0.83
	Existing	Weight sum	1.31 (regardless $\|S_q\|$)			
Cal	Proposed	Weight sum	0.79	1.28	1.57	1.85
		Response time [msec]	1.4	2.3	2.9	3.9
		# of routes	2.27	2.37	2.28	2.25
		Ratio	0.70	0.79	0.85	0.86
	Existing	Weight sum	12.14 (regardless $\|S_q\|$)			

Table 8: Effect of priority queue for various $|S_q|$

Dataset	Approach	2	3	4	5
Tokyo	Proposed	3750	17600	112000	397000
	Distance-based	3890	23500	189000	1760000
NYC	Proposed	13800	108000	172000	637000
	Distance-based	14800	165000	444000	1520000
Cal	Proposed	4900	24800	84900	383000
	Distance-based	5300	34900	168000	899000

If the category sequence size is large, BSSR achieves better performance than PNE and Dij even if we do not use optimization techniques. By comparing Dij to PNE, it can be seen that their performance depends on the datasets and the category sequence size. Although the PNE approach was proposed to be a more sophisticated algorithm than the Dijkstra-based solution [16], PNE requires more time than Dij for the NYC and Cal datasets, which implies that it is not effectively robust to datasets. In terms of RSS, BSSR and PNE achieve nearly the same performance. These two algorithms do not store many routes in the priority queue; therefore, RSS is highly dependent on the graph size. On the other hand, as Dij stores many routes in the priority queue, RSS is significantly larger than those of the other algorithms. Although we do not show the routes returned by each algorithm due to space limitations, all algorithms output the same routes. As a result, BSSR achieves the fastest response time with small memory usage without sacrificing the exactness of the result.

7.3 Optimization Techniques

The optimization techniques improve the efficiency of BSSR. Here, we evaluate each optimization technique.

Initial Search: We show the search spaces with and without an initial search for the first modified Dijkstra algorithm to evaluate the effect of the initial search. Moreover, we evaluate NNinit in terms of response time. Table 7 shows the weight sum, which represents the search space, the response time of NNinit, and the number of sequenced routes found by NNinit for various category sequence sizes. In addition, we show the ratio of the length score of the sequenced route with the largest semantic score among the sequenced routes found in the initial search to the length score of the sequenced route whose semantic score is 0 in the initial search. The weight sum with the initial search is significantly smaller than that without the initial search. We can avoid traversing the whole graph using the initial search; thus, this can significantly reduce the search space of BSSR. Moreover, since the response time of NNinit is significantly less than that of BSSR (Figure 3), we confirm that NNinit can reduce the search space efficiently. Note that the number of sequenced routes found by the initial search is not large. On the other hand, the length score of the sequenced route with the largest semantic score is much smaller than that of the sequenced route whose semantic score is 0. As a result, NNinit reduces the search space significantly without increasing total response time.

Tightening Upper Bound: The priority queue aims at efficiently tightening the upper bound to reduce the search space. Here, we show the total number of vertices visited by BSSR, which is highly related to the response time. Table 8 shows the total number of vertices visited by the proposed priority queue and distance-based priority queue for various category sequence sizes. The number of vertices visited by the proposed priority queue is less than that of the distance-based priority queue. In particular, as the size of the category sequences increases, the performance gap increases because, as the category sequence size increases, the distance-based priority queue cannot find sequenced routes efficiently. Thus, the upper bound is rarely updated. On the other hand, the proposed priority queue can update the upper bound efficiently because the route with the largest size is dequeued preferentially. Thus, the proposed priority queue is more suitable than the distance-based approach for finding SkySRs.

Tightening Lower Bound: To tighten the lower bound, we propose two types of possible minimum distances, i.e., semantic-match and perfect-match minimum distances. If the minimum possible distance is large, we can prune routes even if the routes

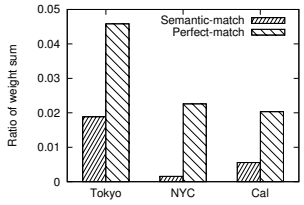
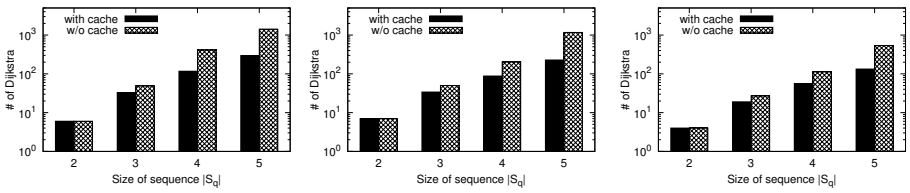


Figure 4: Effect of minimum possible distances



(a) Tokyo **(b) NYC** **(c) Cal**

Figure 5: Effect of on-the-fly caching for various $|S_q|$

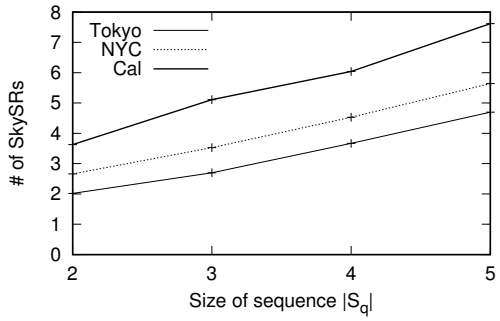


Figure 6: Number of SkySRs for various $|S_q|$

include a small number of PoI vertices. Figure 4 shows the ratios of the possible minimum distances to the sum weights of the initial search when we set the category sequence size to five. The semantic-match and perfect-match minimum distances in the Tokyo dataset effectively reduce the search space by tightening the lower bound. However, different from the Tokyo dataset, the possible minimum distances in the NYC and Cal datasets are small. Since the PoI vertices in the two datasets are relatively concentrated in a small area, the possible minimum distances become small. The effect of the possible minimum distances highly depends on the skews of locations of the PoI vertices.

On-the-fly Caching: On-the-fly caching can reuse the results of former modified Dijkstra algorithm executions; thus, the number of executions of the Dijkstra algorithm decreases. Figure 5 shows the numbers of executions of modified Dijkstra algorithms by BSSR with all optimization techniques and those except for on-the-fly caching. The number of executions of the Dijkstra algorithms decreases using on-the-fly caching. In particular, when the category sequence size increases, the performance gap increases because, as the category sequence size increases, we have more opportunities to reuse former results. Thus, we confirm that on-the-fly caching is effective to reduce the number of executions of the Dijkstra algorithms.

7.4 Number of skyline sequenced routes

Figure 6 shows the number of SkySRs obtained with each dataset for various $|S_q|$. As shown, the Cal dataset returns the largest number of SkySRs. The response time and RSS obtained with the Tokyo and NYC datasets are much greater than the those of the Cal dataset, which implies that the number of SkySRs does not affect response time and RSS significantly. Moreover, if we use a complete real-world dataset, we may not require a ranking function because the number of SkySRs would be small.

Table 9: Example SkySRs in Tokyo

Distance	Sequenced route
7451 meters	Beer Garden → Sushi Restaurant → Sake Bar
1295 meters	Bar → Sushi Restaurant → Sake Bar

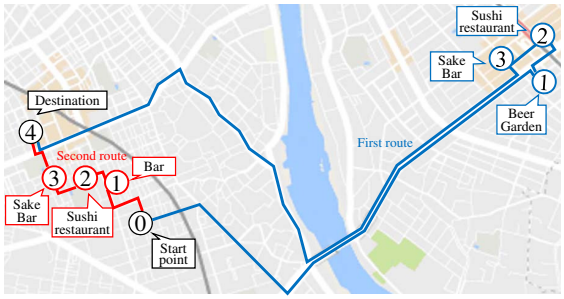


Figure 7: Visualization of routes in Tokyo: black circles (with 0 and 4) denote a start point and a destination, respectively. Blue and red circles denote sequences of PoIs for the first and second routes in Table 9, respectively, and their numbers indicate the order of PoIs to be visited.

7.5 Usecase

We show an example of SkySRs in Tokyo. We assume that we plan to go to places for dinner and drinks. We want to visit a "Beer garden", a "Sushi restaurant", and a "Sake bar" from our current location and finally go to our hotel. Table 9 and Figure 7 show two representative SkySRs from the four identified SkySRs. Note that the other two routes are similar to either of the representative routes. In the Foursquare category trees, "Bar" includes "Beer Garden" and "Sake bar", and "Japanese restaurant" includes "Sushi restaurant". Thus, we find routes using "Bar" and/or "Japanese restaurant". The second route is much shorter than the first route that perfectly matches the user requirement, and the difference between them is only whether they pass a "Bar" or "Beer garden". The best route depends on the users and situations (e.g., weather); thus, we confirm that SkySRs are useful to help users make decisions.

8 USER STUDY

We developed a prototype SkySR query service[6] using OpenStreetMap and the Santander Open Data platform from Santander, Spain[7]. Figure 8 shows a screenshot of the prototype system, which outputs one of the SkySR route. We performed a test in July, 2017. To gather users for this test, the Santander municipality arranged meetings with different groups of people

[6] https://ss.festival.ckp.jp/OuRouteSuggestion/dispSearchRoute/index. The default language is Spanish.
[7] http://datos.santander.es

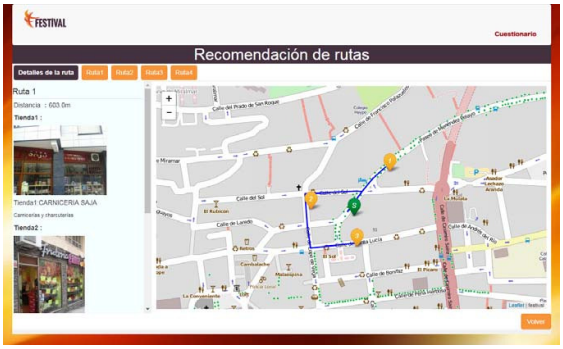


Figure 8: Screenshot of the prototype system

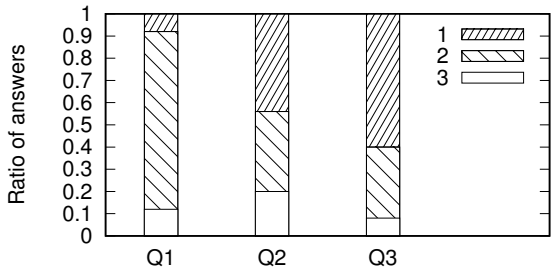


Figure 9: Ratios of answers for each question

to present the service: municipal staff (computing, convention and tourism municipal services), students from vocational training departments who are developing webpages and apps, and citizens. We also provided a leaflet that shows the concept of the SkySR query and how to use the service. In this test, users freely used the service and answered a questionnaire (25 respondents). The questionnaire included the following three questions.

Q1 What do you think about this service?
Answer. 1. I love it, 2. I like it, 3. I do not like it.

Q2 Would you recommend it to anyone?
Answer. 1. Yes, 2. Maybe, 3. No.

Q3 Do you think that it is a good idea for the city: citizens, tourists, commercial sectors?
Answer. 1. Yes, 2. Maybe, 3. No.

We summarize the ratios of answers for each question in Figure 9. As shown, more than 80% of the users liked the service. In addition, the questionnaire shows that the service is valuable for the city. From the user experiment, we confirm that the SkySR query is useful for users and cities.

9 CONCLUSION

In this paper, we have first introduced a semantic hierarchy for trip planning. We then proposed the skyline sequenced route (SkySR) query, which finds all preferred routes from a start point according to a user's PoI requirements. In addition, we have proposed an efficient algorithm for the SkySR query, i.e., BSSR, which simultaneously searches for all SkySRs by a single traversal of a given graph. To optimize the performance of BSSR, we proposed four optimization techniques. We evaluated the proposed approach using real-world datasets and demonstrated that it comprehensively outperforms naive approaches in terms of response time without increasing memory usage or sacrificing the exactness of the result. Moreover, we developed a SkySR query service using open data, and conducted a user test, which confirmed that SkySR queries are useful for both users and cities.

In future work, we would like to extend the proposed approach in several directions. First, because we assume a forest structure for the category classification in this paper, a more complex classification may provide better granularity. Second, because we have not used any preprocessing techniques such as indexing, we plan to propose a suitable preprocessing method for the SkySR query. Finally, although the SkySR query proposed in this paper considers two scores (length and category similarity), it could be extended to consider many attributes of a PoI (e.g., text, keywords, and ratings) and the cost/quality of a graph (e.g., route popularity, tolls, and the number of traffic lights).

ACKNOWLEDGEMENT

This research is partially supported by the Grant-in-Aid for Scientific Research (A)(JP16H01722) and Grant-in-Aid for Young Scientists (B)(JP15K21069).

REFERENCES

[1] Saad Aljubayrin, Zhen He, and Rui Zhang. 2015. Skyline Trips of Multiple POIs Categories. In *DASFAA*. 189–206.
[2] S Börzsöny, Donald Kossmann, and Konrad Stocker. 2001. The Skyline Operator. In *ICDE*. 421–430.
[3] Haiquan Chen, Wei-Shinn Ku, Min-Te Sun, and Roger Zimmermann. 2008. The Multi-rule Partial Sequenced Route Query. In *ACM SIGSPATIAL GIS*. 1–10.
[4] Jian Dai, Chengfei Liu, Jiajie Xu, and Zhiming Ding. 2016. On Personalized and Sequenced Route Planning. *World Wide Web* 19, 4 (2016), 679–705.
[5] Jochen Eisner and Stefan Funke. 2012. Sequenced route queries: Getting things done on the way back home. In *ACM SIGSPATIAL*. 502–505.
[6] Pierre Hansen. 1980. Bicriterion path problems. In *Multiple criteria decision making theory and application*. 109–127.
[7] Xuegang Huang and Christian S Jensen. 2005. In-route skyline querying for location-based services. In *W2GIS*. 120–135.
[8] H-P Kriegel, Matthias Renz, and Matthias Schubert. 2010. Route Skyline Queries: A Multi-preference Path Planning Approach. In *ICDE*. 261–272.
[9] Eugene L Lawler and David E Wood. 1966. Branch-and-bound Methods: A Survey. *Operations research* 14, 4 (1966), 699–719.
[10] Feifei Li, Dihan Cheng, Marios Hadjieleftheriou, George Kollios, and Shang-Hua Teng. 2005. On Trip Planning Queries in Spatial Databases. In *SSTD*. 273–290.
[11] Jing Li, Yin David Yang, and Nikos Mamoulis. 2013. Optimal Route Queries with Arbitrary Order Constraints. *TKDE* 25, 5 (2013), 1097–1110.
[12] Xiaobin Ma, Shashi Shekhar, Hui Xiong, and Pusheng Zhang. 2006. Exploiting a Page-level Upper Bound for Multi-type Nearest Neighbor Queries. In *ACM GIS*. 179–186.
[13] Ernesto Queiros Vieira Martins. 1984. On a multicriteria shortest path problem. *European Journal of Operational Research* 16, 2 (1984), 236–245.
[14] Yutaka Ohsawa, Htoo Htoo, Noboru Sonehara, and Masao Sakauchi. 2012. Sequenced Route Query in Road Network Distance based on Incremental Euclidean Restriction. In *DEXA*. 484–491.
[15] Philip Resnik. 1995. Using Information Content to Evaluate Semantic Similarity in a Taxonomy. In *IJCAI*. 448–453.
[16] Mehdi Sharifzadeh, Mohammad Kolahdouzan, and Cyrus Shahabi. 2008. The Optimal Sequenced Route Query. *The VLDB Journal* 17, 4 (2008), 765–787.
[17] Michael Shekelyan, Gregor Jossé, and Matthias Schubert. 2015. Linear Path Skylines in Multicriteria Networks. In *ICDE*. 459–470.
[18] Yuan Tian, Ken CK Lee, and Wang-Chien Lee. 2009. Finding Skyline Paths in Road Networks. In *ACM SIGSPATIAL GIS*. 444–447.
[19] Zhibiao Wu and Martha Palmer. 1994. Verbs Semantics and Lexical Selection. In *ACL*. 133–138.
[20] Bin Yang, Chenjuan Guo, Christian S Jensen, Manohar Kaul, and Shuo Shang. 2014. Stochastic Skyline Route Planning under Time-varying Uncertainty. In *ICDE*. 136–147.